Nominal Sets in Agda
A Fresh and Immature Mechanization

Miguel Pagano*
FAMAF - Universidad Nacional de Córdoba
Córdoba, Argentina
miguel.pagano@unc.edu.ar

José E. Solsona
Facultad de Ingeniería - Universidad ORT Uruguay
Montevideo, Uruguay
solsona@ort.edu.uy

In this paper we present our current development on a new formalization of nominal sets in Agda. Our first motivation in having another formalization was to understand better nominal sets and to have a playground for testing type systems based on nominal logic. Not surprisingly, we have independently built up the same hierarchy of types leading to nominal sets. We diverge from other formalizations in how to conceive finite permutations: in our formalization a finite permutation is a permutation (i.e. a bijection) whose domain is finite. Finite permutations have different representations, for instance as compositions of transpositions (the predominant in other formalizations) or compositions of disjoint cycles. We prove that these representations are equivalent and use them to normalize (up to composition order of independent transpositions) compositions of transpositions.

1 Introduction

Nominal sets were introduced to Computer Science by Gabbay and Pitts to give an adequate mathematical universe that permits the definition of inductive sets with binding [8]. Instead of taking equivalence classes of inductively defined sets (as in a formal treatment of, say, the Lambda Calculus) or a particular representation of the variables (as in the de Bruijn approach to Lambda Calculus), nominal sets have a notion of name abstraction that ensures all the properties expected for binders; in particular, alpha-equivalent lambda terms are represented by the same element of the nominal set of lambda terms.

In this paper we present a new mechanization [9] of nominal sets. Most of the current mechanizations of nominal sets represent finite permutations as compositions of transpositions, where transpositions are represented by pairs of atoms and compositions as lists. In contrast, our starting point is permutations (i.e. bijective functions); finite permutations are permutations that can be represented by composition of transpositions. Moreover they conflate the set of atoms mentioned in a list with twhe domain of the (represented) permutation. Pondering about this issue, we decided to develop a "normalization" procedure for representations of finite permutations; in order to prove its correctness, we were driven to introduce cycle notation.

The rest of this paper is structured into three sections. In Sect. 2 we summarize the fundamentals of Nominal Sets; then, in Sect. 3 we present the most salient aspects of our mechanization in Agda; and finally in Sect. 4 we conclude by mentioning related works and contrasting them with our approach, indicating also our next steps. We assume some knowledge of Agda, but also hope that the paper can be followed by someone familiar with any other language based on type theory.

*Most of this work was done in a research leave in ORT Uruguay, financed by Agencia Nacional de Investigación e Innovación (ANII) of Uruguay.

Submitted to:
LSFA 2022

2 Fundamentals of Nominal Sets

In this section we summarize the main concepts underlying the notion of Nominal Sets; for a more complete treatment we refer the reader to [11]. We repeat the basic definitions of group and group action. A *group* is a set G with a distinguished element ($\varepsilon \in G$, the *unit*), a binary operation ($\_\cdot\_ \colon G \times G \to G$, the *multiplication*), and a unary operation ($\_^{-1} \colon G \to G$, the *inverse*), satisfying the following axioms:

$$\varepsilon \cdot g = g = g \cdot \varepsilon \qquad g \cdot (g^{-1}) = \varepsilon = g^{-1} \cdot g \qquad g_1 \cdot (g_2 \cdot g_3) = (g_1 \cdot g_2) \cdot g_3$$

Although a group is given by the tuple $(G, \varepsilon, \_\cdot\_, \_^{-1})$ (and the proofs that these operations satisfy the axioms) we will refer to the group simply by G. A sub-group of G is a subset $H \subseteq G$ such that $\varepsilon \in H$ and H is closed under the inverse and multiplication.

Let G be a group. A *G-set* is a set X with an operation $\_\bullet\_ \colon G \times X \to X$ (called the *action*) satisfying:

$$\varepsilon \bullet x = x \qquad g \bullet (g' \bullet x) = (g \cdot g') \bullet x$$

A morphism between G-sets X and Y is a function $F : X \to Y$ that commutes with the actions:

$$F\,(g \bullet x) = g \bullet F\,x$$

These are called *equivariant* functions. Since id_X is equivariant and the composition of equivariant functions yields an equivariant function we can talk of the category of G-Sets.

Any set X can be seen as a G-set by letting $g \bullet x = x$; such a G-set is called the *discrete* G-set. Moreover any group acts on itself by the multiplication.

One can form the (in)finitary product of G-sets by defining the action of g on a tuple in a pointwise manner: $g \bullet \langle t, t' \rangle = \langle g \bullet t, g \bullet t' \rangle$; the projections and the product morphism $\langle F, H \rangle$ are equivariant, assuming that F and H are also equivariant. G-set, as a category, also has co-products.

If X and Y are G-sets one can endow the set Y^X of functions from X to Y with the *conjugate* action: $(g \bullet F)\,x = g \bullet (F\,(g^{-1} \bullet x))$.

G-sets over the Permutation Group The group of symmetries over a set $\mathbb{A}$ consists of $G = \mathit{Sym}(\mathbb{A})$, where $\mathit{Sym}(\mathbb{A})$ is the set of bijections on $\mathbb{A}$; the multiplication of $\mathit{Sym}(\mathbb{A})$ is composition, the inverse is the inverse bijection, and the unit is the identity.

Let $\mathit{Perm}(\mathbb{A})$ be the subset of $\mathit{Sym}(\mathbb{A})$ of bijections that changes only finitely many elements; i.e., $f \in \mathit{Perm}(\mathbb{A})$ if $\mathrm{supp}(f) = \{a \in \mathbb{A} \mid f\,a \neq a\}$ is finite. It is straightforward to prove that $\mathit{Perm}(\mathbb{A})$ is a sub-group of $\mathit{Sym}(\mathbb{A})$. Of course, if $\mathbb{A}$ is finite, then $\mathit{Perm}(\mathbb{A}) = \mathit{Sym}(\mathbb{A})$. Notice that $\mathbb{A}$ is a $\mathit{Perm}(\mathbb{A})$-set with the action being function application: $\pi \bullet a = \pi\,a$. A basic result is that every finite permutation can be decomposed as the composition of *transpositions*: if $a, a' \in \mathbb{A}$, then $(a\ a')$ is the permutation that swaps a with a' and is the identity in $\mathbb{A} \setminus \{a, a'\}$.

Nominal Sets Let X be a $\mathit{Perm}(\mathbb{A})$-set, we say that $x \colon X$ *is supported by* $A \subseteq \mathbb{A}$ if

$$\forall \pi.(\forall a \in A.\pi\,a = a) \implies \pi \bullet x = x \ .$$

We say that X is a *nominal set* if each element of X is supported by some finite subset of $\mathbb{A}$. Since each finite permutation can be decomposed as a composition of transpositions, then one can prove that the above definition is equivalent to

$$\forall a, a' \in \mathbb{A} \setminus A.(a\ a') \bullet x = x \ .$$

In his book [11] Pitts uses classical logic to prove that if x is supported by some finite set A, then there exists a least supporting set, called *the* support of x. As shown by Swan [12] one cannot define the least support in a constructive setting; therefore a formalization in a constructive type theory should ask for "some" finite support. This affects the notion of freshness: in classical logic, x *is fresh for* y if $\mathrm{supp}(x) \cap \mathrm{supp}(y) = \emptyset$, with $x \in X$ and $y \in Y$ being elements of different nominal sets; in a constructive setting one has to limit this relation to atoms: $a \in \mathbb{A}$ is fresh for $x \in X$ if $a \notin \mathrm{supp}(x)$, where $\mathrm{supp}(x)$ is the set supporting x, not necessarily the least one. Notice that the definition is the same ("there exists some finite support for each element"), but in classical logic that is sufficient to obtain the least support.

3 Our Formalization in Agda

Our formalization is developed on top of the Agda's standard library [13]. The standard library includes an algebraic structure going beyond groups; it lacks, however, a formalization of group actions. We present first the definition of Group in the standard library:

```
record Group c ℓ : Set (suc (c ⊔ ℓ)) where
  field
    Carrier : Set c
    _≈_     : Rel Carrier ℓ
    _·_     : Op₂ Carrier
    ε       : Carrier
    _⁻¹     : Op₁ Carrier
    isGroup : IsGroup _≈_ _·_ ε _⁻¹
```

A Group is a *bundle* where the components of its definition (the carrier set, the unit, the inverse, the composition) are explicitly mentioned plus a proof *isGroup* that they satisfy the axioms. Notice that one of the fields is a relation `_≈_`; that relation should be an equivalence relation over the carrier: essentially this amounts to say that the `Carrier` has a setoid structure. Setoids allows for greater flexibility as they enable to work with a notion of equality that is not the propositional equality; `Func A B` is the set of functions between setoids `A` and `B` that preserve the equality.

G-Sets Our first definition is the *structure* that collects the equations required for an action. In the following, we are under a module parameterized by `G : Group`.

```
record IsAction (F : Func (G.setoid ×ₛ A) A) : Set _ where
  _·ₐ_ : Carrier G → Carrier A → Carrier A
  g ·ₐ x = Func.f F (g , x)
  field
    idₐ   : ∀ x → ε ·ₐ x ≈A x
    compₐ : ∀ g' g x → g' ·ₐ g ·ₐ x ≈A (g' · g) ·ₐ x
```

Notice that the record-type `IsAction` is a predicate over functions from the setoid $G \times A$ to A. The definition of GSet is straightforward and follows the pattern of the standard library:

```
record GSet : Set _ where
  field
    set      : Setoid ℓ₁ ℓ₂
    action   : Func (G.setoid ×ₛ set) set
    isAction : IsAction action
```

The next concept is that of equivariant function.[1]

```
record Equivariant  (A : GSet) (B : GSet) : Set _ where
  field
    F : Func (set A) (set B)
    isEquivariant : IsEquivariant (action A) (action B) F
```

Permutations A finite permutation can be given by a bijective map, as a composition of transpositions, or as a composition of disjoint cycles. Let us exhibit this with a concrete example; let $f\colon \mathbb{N} \to \mathbb{N}$ be defined as $f\,x = (x+2)\,mod\,6$ if $x < 6$ and $f\,x = x$ if $x \geqslant 6$; it can be expressed as the composition of two cycles: $(1\,3\,5)\,(0\,2\,4)$. Alternatively, it can also be expressed as composition of transpositions $(1\,3)\,(3\,5)\,(0\,2)\,(2\,4)$.

Transpositions are defined over a decidable setoid; in the following we assume that `A-setoid` has type `DecSetoid ℓ ℓ'`, and `_≟_` decides the equality.

```
  A = Carrier A-setoid ; _≈A_ = _≈_ A ; Perm = Inverse A-setoid A-setoid
  transp : A → A → A → A
  transp a b c with does (c ≟ a)
  ... | true = b
  ... | false with does (c ≟ b)
  ... | true = a
  ... | false = c
```

We started with the following syntactic representation of Finite Permutations:

```
  _≈ₚ_ : Rel Perm _
  F ≈ₚ G = (x : Carrier A) → f F x ≈A f G x
data FinPerm : Set ℓ where
  Id : FinPerm
  Comp : (p q : FinPerm) → FinPerm
  Swap : (a b : A) → FinPerm
```

In order to define the bijection represented by `p : FinPerm` we use `transp` and its properties; here we need to ask the `A-setoid` to be decidable.

```
⟦_⟧ : FinPerm → Perm
⟦ Id ⟧ = idₚ setoid
⟦ Comp p q ⟧ = ⟦ q ⟧ ∘ₚ ⟦ p ⟧ -- _∘ₚ_ is the composition of Perm
⟦ Swap a b ⟧ = record {
    f = transp a b  ; f⁻¹ = transp a b
  ; cong₁ = transp-respects-≈ a b ; cong₂ = transp-respects-≈ a b
  ; inverse = transp-involutive a b , transp-involutive a b }
```

The decidability of the equivalence of `A-setoid` implies the decidability of finite permutations:

```
_≟ₚ_ : ∀ p q → Dec (⟦ p ⟧ ≈ₚ ⟦ q ⟧)
```

The point is that one does not need to check all the atoms but only those in the support of `p`. Moreover we can define a correct "normalization" procedure to get a permutation equal to `p` but without any redundant transposition (and at most one `Id`), passing through a representation by cycles:

[1] We note that Choudhury defines equivariant functions only for the group of finite permutations; it seems absent in the mechanization of Paranhos and Ventura.

```
  norm : FinPerm → FinPerm
  norm = cycles-to-FP ∘ cycles-from-FP
  norm-corr : ∀ p → ⟦ p ⟧ ≈ₚ ⟦ norm p ⟧
```

The functions `cycles-to-FP` maps lists of disjoint cycles to `FinPerm` and `cycles-from-FP` goes in the reverse direction, producing a list of disjoint cycles from a `FinPerm`.

Let us remark that `FinPerm` is just a representation and the set of finite permutation, `PERM`, is the subset of `Perm` corresponding to the image of $[\![\_]\!]$:

```
PERM : Set _
PERM = Σ[ p ∈ Perm ] (Σ[ q ∈ FinPerm ] (p ≈ₚ ⟦ q ⟧))
```

Nominal Sets Remember that a subset $A \subseteq \mathbb{A}$ is a support for x if every permutation fixing every element of A fixes x, through the action. A subset of a setoid `A` can be defined either as a predicate or as pairs (just as in `PERM` where the predicate is `λ p → Σ[ q ∈ FinPerm ] (p ≈ₚ ⟦ q ⟧)`) or as another type, say `B`, together with an injection `ι : Injection B A`.

```
variable
  X : GSet
  P : SetoidPredicate A-setoid
is-supp : Pred X _
is-supp x = (π : PERM) → (predicate P ⊆ _∉-dom (proj₁ π)) → (π ·ₐ x) ≈X x
```

The predicate `λ a → f (proj₁ π) a ≈A a` is `_∉-dom (proj₁ π)`; therefore, if `P a` iff $a \in A$, then `predicate P ⊆ _∉-dom (proj₁ π)` is a correct formalization of $\forall a \in A.\pi\, a = a$.

Our official definition of support is the following:

```
  _supports_ : Pred X _
  _supports_ x = ∀ {a b} → a ∉ₛ P → b ∉ₛ P → SWAP a b ·ₐ x ≈X x
```

Here `SWAP` is a `PERM`utation equal to $[\![$`Swap a b`$]\!]$. We formally proved that both definitions are equivalent.

In order to define nominal sets we need to choose how to say that a subset is finite; as explained by Coquand and Spiwak [7] there are several possibilities for this. We choose the easiest one: a predicate is finite if there is a list that enumerates all the elements satisfying the predicate.

```
finite : Pred (SetoidPredicate setoid) _
finite P = Σ[ as ∈ List Carrier ] (predicate P ⊆ (_∈ as))
```

A GSet is nominal if all the elements of the underlying set are finitely supported.

```
record Nominal (X : GSet) : Set _ where
  field
    sup : ∀ x → Σ[ P ∈ SetoidPredicate setoid ] (finite P × P supports x)
```

It is easy to prove that various constructions are nominals; for instance any discrete GSet is nominal because every element is supported by the empty predicate $\bot_s$:

```
Δ-nominal : (S : Setoid _ _) → Nominal (Δ S)
sup (Δ-nominal S) x = ⊥ₛ , ⊥-finite , (λ _ _ → S-refl {x = x})
  where open Setoid S renaming (refl to S-refl)
```

We have defined `GSet-⇒ A B` corresponding to the GSet of equivariant functions from `A` to `B`; now we can prove that `GSet-⇒ A B` is nominal, again with $\bot_s$ as the support for any `F : Equivariant A B`.

```
  →-nominal : Nominal (GSet-⇒ A B)
  sup (→-nominal) F = ⊥ₛ , ⊥-finite , λ _ _ → supported
    where
    supported : ∀ {a b} x → f ((SWAP a b) ·→ F) x ≈B f F x
```

4 Conclusion

Nominal techniques have been adopted in various developments. We distinguish developments borrowing some concepts from nominal techniques to be applied in specific use cases (e.g. formalization of languages with binders like the λ or π calculus with their associated meta-theory) [2, 6, 5, 4] from more general developments aiming to formalize at least the core aspects of the theory of nominal sets. We are more concerned with the later type.

The nominal datatype package for Isabelle/HOL [14] developed by Urban and Berghofer implements an infrastructure for defining languages involving binders and for reasoning conveniently about alpha-equivalence classes. This Isabelle/HOL package inspired Aydemir et al. [1] to develop a proof of concept for the Coq proof assistant, however it had no further development. In his Master thesis [3], Choudhury notes that none of the previous developments following the theory of nominal sets were based on constructive foundations. He showed that a considerable portion (most of the first four chapters of Pitts book [11]) of the theory of nominal sets can also be developed constructively by giving a formalization in Agda. Pitts original work is based on classical logic, and depends heavily on the existence of the smallest finite support for an element of a nominal set. However, Swan [12] has shown that in general this existence cannot be constructively guaranteed, as it would imply the law of the excluded middle.

Choudhury works with the notion of *some non-unique support*. In order to formalize the category of Nominal Sets, Choudhury preferred setoids instead of postulating functional extensionality. As far as we know, Choudhury is still the most comprehensive mechanization in terms of instances of constructions having a nominal structure.

Recently Paranhos and Ventura [10] presented a constructive formalization in Coq of the core notions of nominal sets: support, freshness and name abstraction. They follow closely Choudhury's work in Agda [3], acknowledging the importance of working with setoids. They claim that by using Coq's type class and setoid rewriting mechanism, much shorter and simpler proofs are achieved, circumventing the "setoid hell" described by Choudhury.

Both of those two formalizations in type theory take a very pragmatic approach to finite permutations: a finite permutation is a list of pairs of names. In our approach, we start with the more general notion of bijective function from which the finite permutations are obtained as a special case; moreover having different representations allowed us to state and prove some theorems that cannot even be stated in the other formalizations. So far, our main contributions are: the representation of finite permutations and the normalization of composition of transpositions; the equivalence between two definitions of the relation "A supports the element x"; and proving that the extension of every container type can be enriched with a group action (notice that this cover lists, trees, etc.).

Our next steps are the definition of freshness, we are studying an alternative notion of support that would admit having a freshness relation between elements of two nominal sets (in contrast with other mechanization that only consider "the atom a is fresh for x") and name abstraction. In parallel we hope to be able to prove that extensions of finite containers on nominal sets are also nominal sets. We also hope to streamline further some rough corners of our development.

Acknowledgments

This formalization grew up from discussions with the group of the research project "Type-checking for a Nominal Type Theory": Maribel Fernández, Nora Szasz, Álvaro Tasistro, and Sebastián Urciouli. We thank Cristian Vay for discussions about group theory. This work was partially funded by Agencia Nacional de Investigación e Innovación (ANII) of Uruguay.

References

[1] Brian E. Aydemir, Aaron Bohannon & Stephanie Weirich (2007): *Nominal Reasoning Techniques in Coq: (Extended Abstract)*. *Electron. Notes Theor. Comput. Sci.* 174(5), pp. 69–77, doi:10.1016/j.entcs.2007.01.028.

[2] Jesper Bengtson & Joachim Parrow (2009): *Formalising the pi-calculus using nominal logic*. *Log. Methods Comput. Sci.* 5(2). Available at `http://arxiv.org/abs/0809.3960`.

[3] Pritam Choudhury (2015): *Constructive Representation of Nominal Sets in Agda*. Master's thesis, Cambridge University.

[4] Ernesto Copello, Nora Szasz & Álvaro Tasistro (2018): *Formalisation in Constructive Type Theory of Barendregt's Variable Convention for Generic Structures with Binders*. *Electronic Proceedings in Theoretical Computer Science* 274.

[5] Ernesto Copello, Nora Szasz & Álvaro Tasistro (2018): *Machine-checked Proof of the Church-Rosser Theorem for the Lambda Calculus Using the Barendregt Variable Convention in Constructive Type Theory*. *Electronic Notes in Theoretical Computer Science* 338, pp. 79–95.

[6] Ernesto Copello, Álvaro Tasistro, Nora Szasz, Ana Bove & Maribel Fernández (2016): *Alpha-Structural Induction and Recursion for the Lambda Calculus in Constructive Type Theory*. *Electronic Notes in Theoretical Computer Science* 323, pp. 109–124. Proceedings of the Tenth Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2015).

[7] Thierry Coquand & Arnaud Spiwack (2010): *Constructively finite?* In: *Contribuciones científicas en honor de Mirian Andrés Gómez*, Universidad de La Rioja, pp. 217–230.

[8] M. J. Gabbay & A. M. Pitts (2002): *A New Approach to Abstract Syntax with Variable Binding*. *Formal Aspects of Computing* 13, pp. 341–363.

[9] Miguel Pagano & José E. Solsona (2022): *Nominal Sets in Agda*. `https://github.com/miguelpagano/nominal-sets/`.

[10] Fabrício S. Paranhos & Daniel Ventura: *Towards a Formalization of Nominal Sets in Coq*. `https://popl22.sigplan.org/details/CoqPL-2022-papers/4/Towards-a-Formalization-of-Nominal-Sets-in-Coq`. Online; accessed 1 May 2022.

[11] Andrew M. Pitts (2013): *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge tracts in Theoretical Computer Science, Cambridge University Press, Cambridge, England.

[12] Andrew Swan (2017): *Some Brouwerian Counterexamples Regarding Nominal Sets in Constructive Set Theory*, doi:10.48550/ARXIV.1702.01556. Available at `https://arxiv.org/abs/1702.01556`.

[13] The Agda Team (2021): *The Agda standard library, version 1.7*. `https://github.com/agda/agda-stdlib`.

[14] Christian Urban & Stefan Berghofer (2006): *A Recursion Combinator for Nominal Datatypes Implemented in Isabelle/HOL*. In Ulrich Furbach & Natarajan Shankar, editors: *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, *Lecture Notes in Computer Science* 4130, Springer, pp. 498–512, doi:10.1007/11814771_41. Available at `https://doi.org/10.1007/11814771_41`.